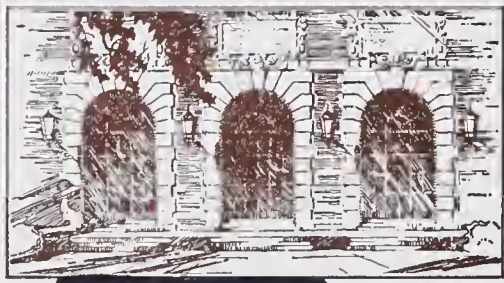


LIBRARY OF THE
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

510.84
I l64
no. 794-799
cop. 2



The person charging this material is responsible for its return to the library from which it was withdrawn on or before the **Latest Date** stamped below.

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

SEP 13 1960



Digitized by the Internet Archive
in 2013

THE PRECOMPILER COMPONENT OF A DATA BASE
DICTIONARY SYSTEM

by

MICHAEL JASON HUGGINS

May 1976



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-76-798

THE PRECOMPILER COMPONENT OF A DATA BASE
DICTIONARY SYSTEM

BY

MICHAEL JASON HUGGINS

May 1976

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*Submitted in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science in the Graduate College of the
University of Illinois at Urbana-Champaign.

ACKNOWLEDGMENT

The author wishes to thank Professor H. G. Friedman and R. L. Mann for their advice and help in the preparation of this thesis.

TABLE OF CONTENTS

INTRODUCTION	1
PART I. PRECOMPILER FUNCTIONAL DESCRIPTION	
Chapter	
I. FUNCTIONAL OVERVIEW	4
II. PRECOMPILER STATEMENTS	8
III. PL/I SOURCE OUTPUT	15
IV. THE COMMUNICATION MODULE	19
V. SUMMARY.	23
PART II. PRECOMPILER INTERNALS AND THE OPERATING ENVIRONMENT	
VI. LEXICAL ANALYSIS	26
VII. SYNTACTIC AND SEMANTIC ANALYSIS.	29
VIII. INTERNAL STRUCTURES.	34
IX. PRECOMPILER OPTIONS.	39
X. PRECOMPILER OPERATING ENVIRONMENT	41
XI. TESTING AND VERIFICATION	43
APPENDIX.	44
REFERENCES	73

LIST OF TABLES

Table	Page
1. FIXED LENGTH AREA IN COMMUNICATION MODULE	20
2. DATA BASE ENTRY IN COMMUNICATION MODULE	21
3. SEGMENT ENTRY IN COMMUNICATION MODULE	21
4. FIELD ENTRY IN COMMUNICATION MODULE	22
5. LEXICAL CLASS ASSIGNMENTS	26
6. INTERNAL HEADER RECORD	35
7. INTERNAL DATA BASE RECORD.	36
8. INTERNAL SEGMENT RECORD	37
9. INTERNAL FIELD RECORD	38
10. PRECOMPILER OPTIONS	40
11. INPUT AND OUTPUT FILES.	42
12. SYSTEM NODE	54
13. PROGRAM NODE	55
14. DATA BASE NODE	56
15. SEGMENT NODE	57
16. FIELD NODE.	59
17. SYSTEM/PROGRAM EDGE	60
18. PROGRAM/DATA BASE EDGE-FIRST PCB ENTRY	61
19. ADDITIONAL PCB ENTRIES	62
20. SENSEG EDGE ENTRY	63
21. DATA BASE/SEGMENT EDGE	64
22. SEGMENT/FIELD EDGE	66

LIST OF TABLES--Continued

Table	Page
23. SAMPLE HOJ RECORD	67
24. RANDOMIZING MODULE HOJ DATA	68
25. EDIT/VERIFICATION HOJ DATA	68
26. XDFIELD HOJ DATA.	69
27. DATA SET GROUP HOJ DATA	70
28. LOGICAL CHILD HOJ DATA	71
29. LAT RECORD.	72

LIST OF FIGURES

Figure	Page
A. PCB declarative	10
B. Data base declarative	11
C. Segment declarative	12
D. Field declarative	13
E. Segment manipulation	14
F. Nonsegment manipulation	14
G. PCB, data base, and segment sample output	16
H. Data manipulation sample output	18
I. Lexical analysis	28
J. Model syntax table	30
K. Data manipulation syntax automaton	33
L. Logical view of dictionary system	48
M. Example of segment-field edge	49
N. Logical view of HOJ table	50
O. Logical view of LAT table	51

INTRODUCTION

With the advent of large, general purpose data base systems [1], several desirable information processing theories have now been implemented. These include advances in the areas of data independence, data sharing, data security, and control. While facilities to take advantage of these concepts have been implemented to varying degrees, much of the control needed to administer their use is not inherent in the data base software itself. To meet this need, the role of data base administration has emerged [2]. While data base administration is finding its place in data processing structures, much work is being done to provide it with the tools needed to manage and control the data. The greatest need is in the area of data dictionaries.

A data dictionary is a collection of data about data [3]. A complete description of a particular installation's data base environment would be contained within the data dictionary. The use of a dictionary provides a large measure of control and documentation which allows data sharing and security to be used and monitored. The real drawback is that the dictionary, while an excellent source of information and an aid in communicating with the data base software, does not actually control the access to the data. If the dictionary were the source of information controlling the actual interface between a program and the data it wished to process, then a real level of data independence and security could be provided, and many additional services could be made available. To this end the Data Base Dictionary System described in Appendix A was designed.

This project is a sample implementation of one subsystem of the Data Base Dictionary System, the precompiler. IBM's Information Management System [4] (IMS) is used here as the target Data Base Management System (DBMS) because it is general purpose and is currently in use by many installations. The precompiler is an extension to PL/I and is used to generate IMS application programs. In addition to simplified programming, the goals of the precompiler are to implement the various features offered by the Data Base Dictionary System described in Appendix A. Until such time when data dictionaries, data base software and compilers are closely associated and more fully integrated, a precompiler can be most useful in bridging the gap between these separate systems and providing the needed support to the application programmer.

PART I
PRECOMPILER FUNCTIONAL DESCRIPTION

CHAPTER I

FUNCTIONAL OVERVIEW

The function of this precompiler is to take as input an application program written in PL/I with the addition of certain precompiler statements and generate a complete PL/I source program along with an interface module for use by the execution monitor. In this role, the precompiler serves not only as a programming aid but also as the first level of security and control in the Data Base Dictionary System environment.

The precompiler statements fall into two categories, declaratives and data manipulation statements. The declaratives allow for the declaration of Program Control Blocks [5] (PCB), data bases, and logical segments for which the appropriate PL/I DECLARE statements are generated. In addition to these declaratives there are a set of precompiler statements used to communicate requests to the execution monitor. These data manipulation statements generate PL/I source code which includes a CALL to RNPTDLI, the execution monitor.

The concept of the logical segment is essential to many of the features that the dictionary system offers. In effect, it is the logical segment approach that allows for field level independence not inherent in IMS. The major concepts surrounding the logical segment are as follows:

1. Any field may be included in the logical segment as long as it is contained in the real or source segment.

That is, a logical segment is a subset of its source segment.

2. A field may be requested in any scale, base or precision. Conversion will be accomplished by the execution monitor based on information from the dictionary and the communication module.
3. Field position within the logical segment is totally independent of its real location within the source segment. Again the execution monitor performs the necessary mapping at run-time.
4. A program may update a subset of a real segment without affecting fields in the source segment that it is not sensitive to.

The logical segment approach allows for run-time binding. The execution monitor establishes the mappings and conversions necessary to give the program the data it requests. With the precompiler and execution monitor functioning together in this manner, true data independence is achieved. As long as the data fields requested remain in the source segment, all other rearrangements and format changes are transparent to the program and do not require a recompile or relink edit.

The precompiler checks security at several levels. Because this precompilation is the first security check and therefore a requirement, a method has been devised to ensure that a program executing in the Data Base Dictionary System environment has been processed by the precompiler. As the program is processed, the following is ensured:

1. The program is described to the dictionary system and is written in PL/I.
2. The data bases requested are in the system indicated. If the system is password protected, then the password is given by the program.
3. The program is allowed to access each data base it requests.
4. The program is sensitive to the segments it requests and update access is allowed if attempted.
5. The program is allowed the type of access requested to each field within the logical segments defined.

In the area of simplified programming, several precompiler features make the task of creating a complete application program easier. The PCB mask, a moderately large structure, is generated for each PCB declared in the program. On the precompiler statements themselves, several options are inferred if not explicitly stated. If the program wishes to process a segment exactly as it is in the data base, then the precompiler will generate the appropriate structure to map the requested segment without program concern for the declaration of all the associated fields. The data manipulation specifications expand into the necessary source statements including the CALL to interface with the execution monitor and IMS.

In addition to these "shorthand" techniques, a programmer using the dictionary system need not worry as much about data editing,

segment characteristics, and data conversion. This means that he can concentrate on the function to be performed. While allowing the programmer to accomplish his task more efficiently, the precompiler as a part of the dictionary system adds a real measure of data independence and control to a processing environment.

CHAPTER II

PRECOMPILER STATEMENTS

Precompiler statement syntax is similar to PL/I in that it is keyword oriented. Data associated with a keyword is enclosed in parenthesis following that word. A set of related keywords is expressed as a precompiler statement. The semicolon is used as the statement terminator. The period immediately followed by either "DECLARE" or one of the IMS function abbreviations [5] signals the beginning of a precompiler statement. Within the context of a statement, the keywords are treated as reserved words and therefore cannot be used as user symbols. These reserved words are: ASIS, BASED, BIN, CHAR, DATABASE, DEC, FIELDS, FIXED, FLOAT, KEYFDBKLEN, NAME, PCB, PROCOPT, SEGMENT, SOURCE, SSA, SYSTEM, and WITH.

Syntax conventions are again much like those of PL/I. The precompiler is blank transparent, that is, any number of consecutive blanks are treated only as a token separator. Except as a token separator, card boundaries and comments are also transparent. Quoted strings are treated as one token regardless of their content.

The precompiler scans the input program looking for one of its statements. When one is found, it is processed token by token until the semicolon is found. If an error is detected, the remainder of the statement in error is skipped. When the precompiler has finished parsing one of its statements, scanning continues until another

is found or end of file is reached. Each precompiler statement must begin a PL/I statement, or in the case of data manipulation requests, be the only entry in a THEN or ELSE clause of an IF statement.

There are three types of declarative statements and twelve data manipulation statements. Each declarative must begin with the token ".DECLARE". Data manipulation statements also begin with a period immediately preceding one of the following IMS function abbreviations [5]: GU, GN, GNP, GHU, GHN, GHNP, ISRT, DLET, REPL, SNAP, CHKP, LOG.

A detailed definition of the syntax of each precompiler statement and the semantic action taken in each case is shown in Figures A through F. In all cases the keywords may appear in any order but only once per statement. The notation conventions used in these figures to describe the syntax are as follows:

1. Nonterminals are enclosed in braces and explained below each use.
2. Items enclosed in plain brackets are optional.
3. Items enclosed in brackets followed by a superscript "+" are optional and may be repeated any number of times.
4. Parentheses are terminals and must be included where indicated.
5. The bar separates a list from which one and only one item must be chosen.
6. User variables, passwords, and SSA names follow standard PL/I conventions for symbol formation.

SYNTAX:

```
.DECLARE PCB NAME (<id>) BASED (<id>) KEYFDBKLEN (<num>) ;
```

where

<id> is a user variable

<num> is an unsigned decimal integer

SEMANTIC ACTION:

1. establish this as the current PCB
2. allocate an internal PCB entry and save the pertinent information
3. output the PL/I structure to map this PCB

ERROR CONDITIONS:

1. invalid syntax
2. PCB already known

Fig. A.--PCB declarative

SYNTAX:

```
.DECLARE DATABASE NAME (<id>) SYSTEM (<id>[,<pswd>])
    [PCB (<id>)];
```

where

<id> is a user variable

<pswd> is the password associated with the system

SEMANTIC ACTION:

1. associate the data base with the indicated PCB, or if the PCB is not specified, with the current PCB
2. verify data with dictionary
3. if PCB is specified, make it the current PCB

ERROR CONDITIONS:

1. invalid syntax
2. data base not known to the dictionary
3. program not allowed to access this data base
4. system not known to the dictionary, or if passworded, the password given does not match
5. PCB not known, or if no PCB specified, no current PCB
6. PCB already associated with a data base

Fig. B.--Data base declarative

SYNTAX:

```
.DECLARE SEGMENT NAME (<id>) [ASIS] [PCB (<id>)]
    [SOURCE (<id>)] [PROCOPT (<procopt>)]
    [ [WITH] FIELDS <field-declarative>[,<field-declarative>]+];
```

where

<id> is a user variable

<procopt> is a valid IMS processing option [5]

<field-declarative> is defined in Figure D

note

all keywords must precede the field declaratives, if any

SEMANTIC ACTION:

1. allocate an internal segment entry and save the pertinent information
2. identify the source segment, either explicitly or implicitly
3. if PCB is specified, establish it as the current PCB
4. if ASIS is specified, generate the field entries for this segment as it is defined to the dictionary
5. check security, i.e. the program's access to this segment
6. output the PL/I structure to map this segment

ERROR CONDITIONS:

1. invalid syntax
2. source segment not known to the dictionary
3. logical segment already declared
4. invalid processing option
5. PCB not known, or if no PCB is specified, no current PCB
6. source segment not in data base
7. program not allowed access to this segment
8. in the ASIS case, the program is not allowed to access one or more fields in the source segment

Fig. C.--Segment declarative

SYNTAX:

<id> CHAR|DEC|BIN|ZONED (<len>[,<num>]) [FIXED|FLOAT]

where

- <id> is a user type symbol which is a field name
- <len> an unsigned integer representing the total field length
- <num> an unsigned integer representing the number of decimal places

SEMANTIC ACTION:

1. allocate an internal field entry and save the pertinent information
2. check security
3. include this field in the structure mapping the current segment
4. keep track of maximum segment size

ERROR CONDITIONS:

1. invalid syntax
2. invalid scale, base or precision combination
3. field not known to the dictionary
4. program not allowed the requested level of access to the field
5. field is not in the source segment being defined

Fig. D.--Field declarative

SYNTAX:

.<func> <seg> [SSA (<id>],<id>]⁺);

where

<func> is either GU, GN, GNP, GHU, GHN, GHNP, ISRT, DLET
or REPL

<seg> is a user type symbol which is a segment name

<id> is a user variable

SEMANTIC ACTION:

1. output CALL and preliminary set up statements
2. keep track of the maximum number of SSAs in any one call

ERROR CONDITIONS:

1. invalid syntax
2. segment not known
3. invalid use of segment

Fig.E.--Segment manipulation

SYNTAX:

.<func> <area> PCB (<id>) ;

where

<func> is either SNAP, CHKP or LOG

<area> is a user variable

<id> is a user variable

SEMANTIC ACTION:

1. output CALL and preliminary set up statements

ERROR CONDITIONS:

1. invalid syntax
2. PCB is not known

Fig. F.--Nonsegment manipulation

CHAPTER III

PL/I SOURCE OUTPUT

The source code produced is a complete PL/I program ready to be compiled. All the precompiler statements are included in it as comments, followed by the appropriate generated code. The expanded code can take at least forty-five characters per line. If the margin length as defined by the compiler option MARGINS is not at least forty-five characters, then precompilation is abandoned.

The `".DECLARE PCB"` statement is expanded into the PCB mask necessary to map the control blocks passed to each program by IMS. A detailed description of each element in the structure can be found in [5]. Figure G shows the precompiler statement converted to a comment followed by the expanded PCB mask. The `".DECLARE DATABASE"` statement does not result in any PL/I code but is included as a comment as also illustrated in Figure G.

The `".DECLARE SEGMENT"` precompiler statement is expanded into an unaligned structure that maps the logical segment as defined in the program or the real segment if the ASIS option is taken. Figure G shows how the precompiler statement is turned into a comment and followed by the appropriate structure for use as an I/O area.

The data manipulation precompiler statements are treated much the same as the declaratives. They are included as a comment within the generated program, followed by the necessary PL/I source code to

```

SAMPDATA:  PROCEDURE OPTIONS(MAIN);                                00000010
  DECLARE REGULAR-DATA CHAR(9) INIT(''.DECLARE''),                00000020
    SOMEBITS BIT(5);                                              00000030
/*****
  .DECLARE PCB NAME(PCBONE) BASED(PTR1) KEYFDBKLEN(6);            00000031
*****/00000033
  DCL PTR1 POINTER;                                              00000034
  DCL 1 PCBONE  BASED(PTR1),                                     00000035
    5 DBD_NAME CHAR(8),                                         00000036
    5 SEG_LEVEL CHAR(2),                                         00000037
    5 STATUS_CODE CHAR(2),                                       00000038
    5 PROC_OPTIONS CHAR(4),                                       00000039
    5 RESDLI FIXED BIN(31),                                       00000040
    5 SEG_NAME CHAR(8),                                           00000041
    5 LEN_KFDBK FIXED BIN(31),                                    00000042
    5 NUM_SENSEGS FIXED BIN(31),                                   00000043
    5 KEY_FDBK_AREA CHAR(6);                                       00000044
                                                                00000045
/*****
  .DECLARE DATABASE NAME(SAMPDATA) SYSTEM(SAMPDATA,TSTPSWD);    00000047
*****/00000048
                                                                00000050
/*****
  .DECLARE SEGMENT NAME(LOGSEG1) SOURCE(SAMPSEG) PCB(PCBONE)    00000051
    PROCOPT(AP) WITH FIELDS                                       00000070
      SAMPFLD1 CHAR(5),                                           00000080
      SAMPFLD2 ZONED(8,2),                                         00000090
      SAMPFLD3 FIXED DEC(10,3),                                    00000100
      SAMPFLD4 FIXED BIN(31),                                       00000110
      SAMPFLD5 FLOAT DEC(6);                                       00000111
*****/00000112
  DCL 1 LOGSEG1  UNALIGNED,                                       00000113
    5 SAMPFLD1 CHAR(5),                                           00000114
    5 SAMPFLD2 PIC '999999V9T',                                    00000115
    5 SAMPFLD3 FIXED DEC(10,3),                                    00000116
    5 SAMPFLD4 FIXED BIN(31),                                       00000117
    5 SAMPFLD5 FLOAT DEC(6);                                       00000118
  DCL FUNCTION CHAR(4),                                           00000119
    PARMCOUNT FIXED BIN(31);                                       00000120
                                                                00000121
SOMEBITS = '01010'B;                                           00000130
END SAMPDATA;                                                    00000140

```

Fig. G.--PCB, data base, and segment sample output

interface with the execution monitor. If the precompiler request is coded as the only entry in an IF-THEN or IF-ELSE clause, then a DO group is created containing the generated code. This maintains the intended program structure.

Figure H shows how the data manipulation statements are handled. The code produced sets a variable to the proper number of parameters in the CALL to follow and finally invokes the execution monitor with the proper parameters. The two variables FUNCTION and PARMCOUNT are declared and maintained by the precompiler and therefore do not require programmer concern. Standard IMS Segment Search Arguments (SSA) [5] are used and when included in a precompiler request statement, are passed to IMS by the execution monitor.

```

SAMPDATA:  PROCEDURE OPTIONS(MAIN);                                00000010
  DECLARE SSA1 CHAR(8) INIT('SGEFPAMS'),                          00000020
         SSA2 CHAR(8) INIT('SGEFPANM'),                          00000030
         LOGRECORD CHAR(12); .                                    00000040
/*****
  .DECLARE PCB NAME(PCBONE) BASED(PTR1) KEYFDBKLEN(8);            00000041
*****/00000042
  DCL PTR1 POINTER;                                              00000044
  DCL 1 PCBONE  BASED(PTR1),                                     00000045
        5 DBD_NAME CHAR(8),                                     00000046
        5 SEG_LEVEL CHAR(2),                                    00000047
        5 STATUS_CODE CHAR(2),                                  00000048
        5 PROC_OPTIONS CHAR(4),                                 00000049
        5 RESDLI FIXED BIN(31),                                 00000050
        5 SEG_NAME CHAR(8),                                     00000051
        5 LEN_KFDBK FIXED BIN(31),                             00000052
        5 NUM_SENSEGS FIXED BIN(31),                           00000053
        5 KEY_FDBK_AREA CHAR(8);                                00000054
                                                                00000055
/*****00000056
  .DECLARE DATABASE NAME(SAMPDATA) SYSTEM(SAMPDATA,TSTPSWD);    00000057
*****/00000058
                                                                00000060
/*****00000061
  .DECLARE SEGMENT NAME(LOGSEG1) SOURCE(SAMPSEG) PCB(PCBONE)    00000070
        PROCOPT(A) WITH FIELDS                                   00000080
        SAMPFLD FIXED DEC(6,3);                                 00000081
*****/00000082
  DCL 1 LOGSEG1 UNALIGNED,                                       00000083
        5 SAMPFLD  FIXED DEC (6,3);                             00000084
  DCL FUNCTION CHAR(4),                                          00000085
        PARMCOUNT FIXED BIN(31);                              00000086
                                                                00000090
/**** PROCESSING FOLLOWS ****/                                  00000100
LOGRECORD = 'SAMPLE LOG';                                       00000110
/**** SEGMENT MANIPULATION STATEMENT FOLLOWS ****/            00000120
/*****00000121
  .GU LOGSEG1 SSA(SSA1,SSA2);                                   00000122
*****/00000123
  FUNCTION = 'GU ' ;      PARMCOUNT = 3 + 2;                  00000124
  CALL RNPTDLI(PARMCOUNT,                                     00000125
        FUNCTION,                                             00000126
        PCBONE,                                              00000127
        LOGSEG1,                                             00000128
        SSA1,                                                00000129
        SSA2);                                              00000130
                                                                00000131
/**** NON-SEGMENT MANIPULATION STATEMENT FOLLOWS ****/        00000140
/*****00000141
  .LOG LOGRECORD PCB(PCBONE);                                  00000142
*****/00000143
  FUNCTION = 'LOG ' ;      PARMCOUNT = 3 ;                   00000144
  CALL RNPTDLI(PARMCOUNT,                                     00000145
        FUNCTION,                                             00000146
        PCBONE,                                              00000147
        LOGRECORD);                                          00000148
                                                                00000150
END SAMPDATA;                                                  00000160

```

Fig. H.--Data manipulation sample output

CHAPTER IV

THE COMMUNICATION MODULE

After the input program has been processed, a communication section in the form of an object module is produced. This module contains both executable code and a tabular description of each data base, each logical segment, and each field declared within the program. The application program object module produced by the regular PL/I compiler is linked with this module to become the complete executable application program. At run time, the execution monitor will load the application program and modify some code within the communication module to allow it to be the entry point to which IMS will transfer control. In addition, some address references will be linked such that the application program can communicate with the execution monitor. With the description of the program's data requirements contained within the communication module, and the actual segment descriptions from the dictionary, the execution monitor is able to determine the necessary mappings and data conversions.

The tabular section of the communication module is composed of three subsections: the data base section, the segment section, and the field section. The data base section contains one entry for each data base declared within the program. Similarly, the segment and field sections contain one entry for each segment or field respectively. These three subsections are preceded by a fixed length area containing

the executable code and some control information. The layout of each subsection is described in the following tables.

TABLE 1
FIXED LENGTH AREA IN COMMUNICATION MODULE

Decimal Dis- placement	Field Size	Data Format	Content
0	108	Code	Executable code which contains the program entry point (RNENTRY) and the interface point (RNPTDLI) back to the execution monitor
108	2	Binary	The maximum number of SSA's used in any CALL to RNPTDLI within the program
110	2	Binary	The CSECT size less the 114 bytes in the fixed length area, but at least as large as the largest segment
112	2	Binary	The number of data bases declared in the program

TABLE 2
DATA BASE ENTRY IN COMMUNICATION MODULE

Decimal Dis- placement	Field Size	Data Format	Content
0	8	Character	The data base name
8	2	Binary	The number of segments in the data base
10	2	Binary	The offset to the first segment entry for the segments in this data base relative to the beginning of this entry

TABLE 3
SEGMENT ENTRY IN COMMUNICATION MODULE

Decimal Dis- placement	Field Size	Data Format	Content
0	8	Character	The logical segment name used in the pro- gram for this segment
8	8	Character	The real segment which is the source for this logical segment
16	2	Binary	The number of fields in this segment
18	2	Binary	The offset to the first field entry for the fields in this segment relative to the beginning of this entry

TABLE 4
FIELD ENTRY IN COMMUNICATION MODULE

Decimal Dis- placement	Field Size	Data Format	Content
0	8	Character	Field name
8	2	Binary	The field length in bytes minus one
10	1	Bit string	An indication as to the data type of this field. Possible codes: Bit 0=1 FLOAT Bit 1=1 FIXED Bit 2=1 CHAR Bit 3=1 PACKED Bit 4=1 ZONED Bit 5=1 /SX Bit 6=1 /CK Bit 7=1 XDFIELD Bits 0-7=1 HEX
11	1	Binary	Field scale factor
12	2	Binary	Field position as an offset into logical segment

CHAPTER V

SUMMARY

The precompiler subsystem of the Data Base Dictionary System can be briefly summarized as follows. As an extension to PL/I it allows several features of the Data Base Dictionary System to be implemented. Since programs are processed before they are actually compiled and the dictionary is available at this precompile time, security and access control is enforced and programming simplification is provided for. Using a set of precompiler statements, a program declares its intentions regarding what data and how that data is to be processed. Authority and continuity is checked. Requests for data are coded using another set of precompiler statements.

The concept of the logical segment is perhaps the most important technique employed. A logical segment is a subset of a real segment of data as defined to IMS. Any of the data elements within the source segment may be requested in any order and in any format. Conversions and mappings will be done by the execution monitor. Chapter I examines the advantages of this approach.

Physically then, the precompiler reads in a program which includes the special statements, processes the program accessing the dictionary as needed, and optionally produces a listing of the input, a listing of the expanded program produced, the expanded program for input to the PL/I compiler, the expanded program for punching, a

communication module for interface with the execution monitor, and always a burst page and statistics. Chapter X gives a more detailed description of the processing options available. Although this precompilation process requires an extra step in the translation from source program to executable code, the benefits gained are worth the additional overhead.

PART II

PRECOMPILER INTERNALS AND THE OPERATING ENVIRONMENT

CHAPTER VI

LEXICAL ANALYSIS

Lexical analysis may be defined as the scanning of the characters in a source program from left to right isolating tokens or symbols. A scanner is used in this precompiler to perform lexical analysis as well as to determine the type of token isolated. When the syntactic and semantic routines invoke the scanner, the next token is found and its type made available. To perform the analysis, each character is translated into a lexical class as defined in Table 5.

TABLE 5
LEXICAL CLASS ASSIGNMENTS

Class	Members
Blanks	Blanks
Letters	A thru Z and #, \$, @
Underscore	␣
Quote	"
Digits	0 thru 9
Delimiters	. + - = % & ; () , : < > ␣
Double	".DECLARE" or a .<func> as previously defined
Slash	/
Star	*
Bad Character	Anything else

While scanning the input program quoted strings are treated as a single token without regard to the characters within that string.

Except for terminating tokens, card boundaries and comments are ignored. Quoted strings, however, may extend onto multiple cards. When a token has been isolated, its class is determined by searching a table of possible token types. If the token in hand is not in the table, then it is an "undefined" token and is so classed. The possible classes of token types are undefined tokens, numbers, strings, IMS functions, delimiters, and reserved words. For ease of reference, the scanner not only indicates which token class is isolated but also, when applicable, which IMS function, delimiter or reserved word. If while scanning the program the end of the source is found, an indication of such is given by the scanner so that the syntactic routines can take appropriate action.

Figure I shows a flow of the lexical analysis process. When the lexical analysis routine is invoked, processing starts at the "ENTER" node. Each "NEXT CHAR" box represents moving to the character at the right of the current position. From each box extends one or more flow lines indicating the action taken based on the particular lexical class of the current character. Some lines are followed for several classes. Those lines with no class indicated show action taken for lexical classes not explicitly covered by other lines. "RETURN" means that a token has been isolated and typed and control has passed to the invoking routine. The reader should recall that card boundaries do terminate tokens (except quoted strings) but are transparent otherwise.

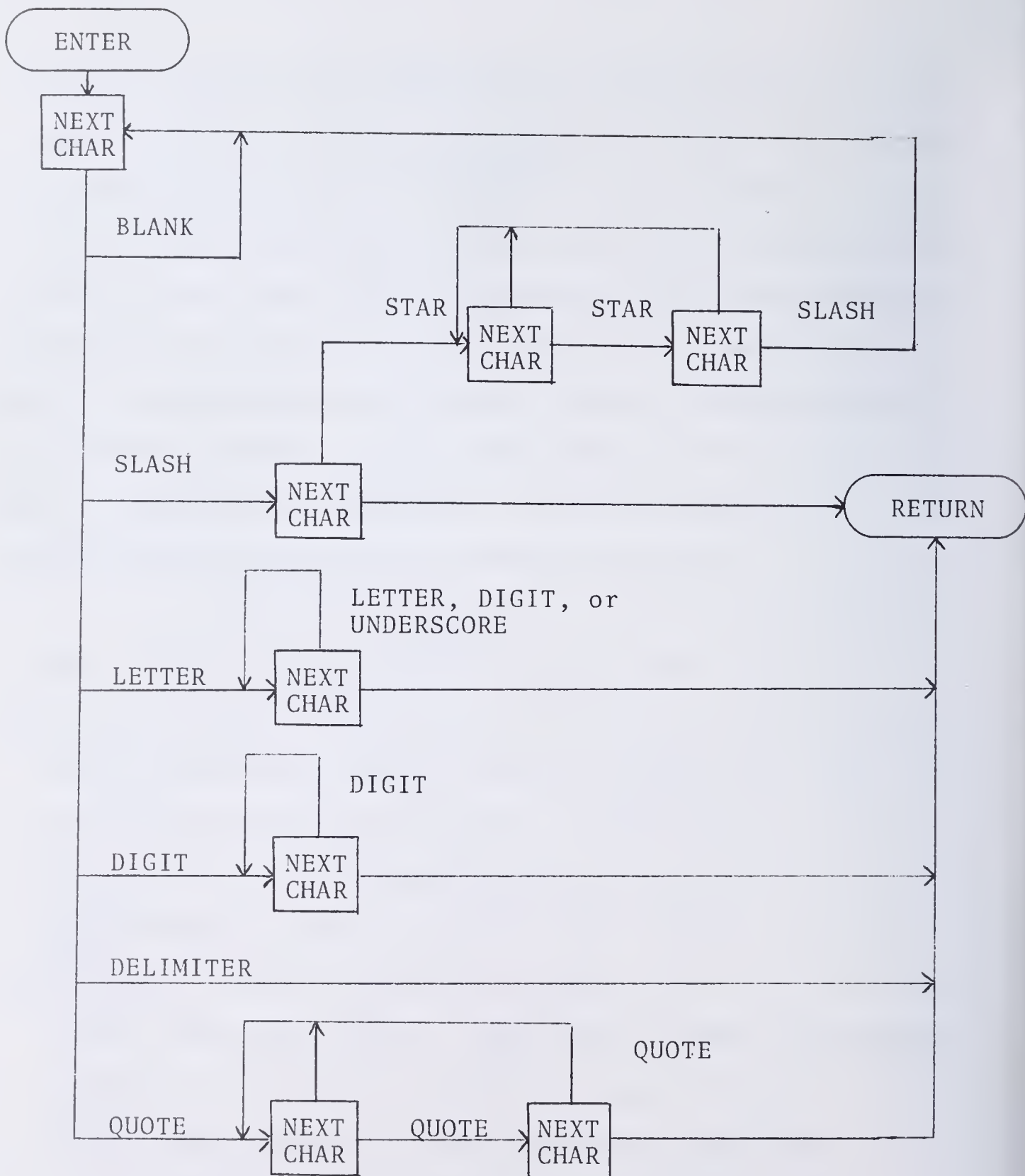


Fig. I.--Lexical analysis

CHAPTER VII

SYNTACTIC AND SEMANTIC ANALYSIS

The syntactic analysis or parsing of the input source program is performed at two levels, the outermost of which performs two functions. First, the source program is parsed until the program name is found. Since this is a PL/I program, its name should be the token preceding the first colon, i.e., the label on the first external procedure. Once the program name is found, the dictionary is checked to verify that the program is defined and that it is written in PL/I. As with IMS alone, programs must be defined before they can be used. If a discrepancy exists, an appropriate error message is produced and precompilation is abandoned.

The second function performed is the search for a precompiler statement. Precompiler statements are divided into their two semantic classes. Once one of the tokens identifying a precompiler statement has been found, control is passed to one of two routines, one for the declarative and the other for the data manipulation statements. One of these two routines then performs the second level of syntactic analysis.

The declarative routine initially ensures that the precompiler statement about to be parsed starts a new statement in the input program. Parsing then is accomplished by moving through a series of parse tables that are linked together in such a way that syntactical analysis and semantic processing are performed quite simply. Figure J shows the

NUMBER OF ENTRIES		
TOKEN VALUE	PROCESSING ROUTINE	NEXT TABLE
-1	BAD SYNTAX ROUTINE	NEXT TABLE

Fig. J.--Model syntax table

structure of these tables. After the initial table is established the parser repeats the following:

1. The lexical analysis routine is called to get the next token.
2. The current syntax table is searched for the entry that corresponds to the current token.
3. The indicated routine is called to take semantic action based on the token in hand.
4. The table indicated as the next table becomes the current table.

This process is continued until a statement terminator, the semicolon, is found or the end of the input program is reached. The semantic

routines invoked process a particular keyword and its operands, if any. Processing includes accessing the dictionary for verification and security functions as well as maintenance of the internal structures. A full description of these structures is found in Chapter VIII of this document.

When the outermost level of the syntactic parser finds a data manipulation statement, the second inner level routine is invoked. Since the semantic action required for this type of statement is much less than that required for declarative statements, a finite state automaton approach was used to parse them. This technique affords good syntax analysis while supporting the limited semantic processing required. No dictionary access is needed. If the request is a data base manipulation function then the I/O area given must be a logical segment that has been previously defined. If, however, the function is SNAP, CHKP or LOG then the PCB given must have been previously defined.

Figure K graphically illustrates the nine state automaton and the movement through the states for different types of expected input. Parsing begins at START after the I/O area has been identified. If an unexpected token is found, then the processing of this statement is terminated. Four cases of missing right parentheses are shown with dotted lines. In these cases the missing token is assumed.

While the outer level routine identifies the precompiler statements and invokes the appropriate second level routine, syntactic and semantic analysis continues until the end of the input program is

reached. When errors are detected error messages are produced and the remainder of the current statement is bypassed.

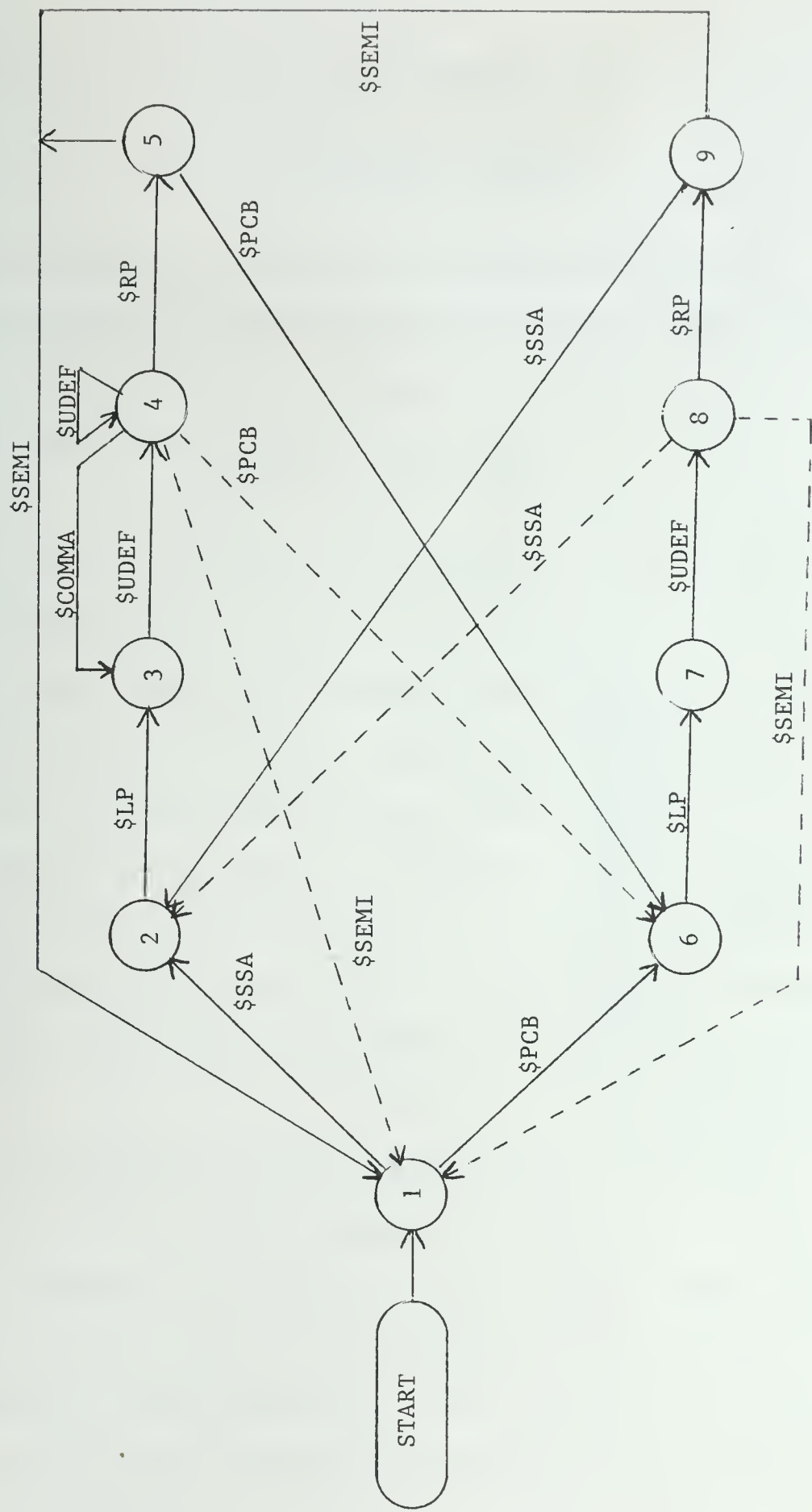


Fig. K.--Data manipulation syntax automaton

CHAPTER VIII

INTERNAL STRUCTURES

A set of internal tables is created and maintained throughout the precompilation process. These tables contain the information necessary to verify the correctness of and the continuity between the entities declared by the program. In addition, they accumulate data used to generate the communication module. There are four table types. The first is a header record that contains counters and other static variables as well as the heads of the linked lists connecting the other tables types. The second, third and fourth table types represent each data base, segment, and field declared respectively. From the header record, all the data base table occurrences are linked on a list. All segment table occurrences are linked on a second list. Each data base table contains the head of a list of segment tables that represent the segments within that data base. In like manner, each segment table contains the head of a list of field tables for the fields contained within that segment.

This network of interrelated tables is built from the precompiler declarative statements and information from the dictionary. As each new statement is being processed, the current environment depicted by these internal tables is checked to see if the new entity fits in. If it does then the necessary tables are created and/or maintained. When processing the data manipulation statements, the tables are checked

to ensure the feasibility of the request in hand. When the entire input source program has been processed, the tables are used to create the communication module. It in turn is linked with the object module from the PL/I compiler to form the complete application program. The layout of each of the four internal records is shown in Tables 6 through 9.

TABLE 6
INTERNAL HEADER RECORD

Decimal Dis- placement	Field Size	Data Format	Content
0	2	Binary	Number of data bases declared
2	2	Binary	Number of segments declared
4	2	Binary	Number of fields declared
6	2	Binary	Maximum number of SSAs used in any CALL statement
8	2	Binary	The size of the largest segment
10	4	Pointer	Head of the linked list of data base records
14	4	Pointer	Head of the linked list of segment records

TABLE 7
INTERNAL DATA BASE RECORD

Decimal Dis- placement	Field Size	Data Format	Content
0	2	Binary	The eventual location of the corresponding data base entry in the communication module
2	4	Pointer	Link to next data base record
6	8	Character	Data base name
14	8	Character	System name
22	8	Character	PCB name
30	2	Binary	Number of segments in this data base
32	4	Pointer	Pointer to first segment record for this data base

TABLE 8
INTERNAL SEGMENT RECORD

Decimal Dis- placement	Field Size	Data Format	Content
0	2	Binary	The eventual location of the corresponding segment entry in the communication module
2	4	Pointer	The link to the next segment record from the header record
6	8	Character	Logical segment name
14	8	Character	Source segment name
22	8	Character	PCB name
30	4	Character	The PROCOPT for this segment
34	2	Binary	Number of fields in this segment
36	4	Pointer	Link to next segment in its data base
40	4	Pointer	Link to first field in this segment

TABLE 9
INTERNAL FIELD RECORD

Decimal Dis- placement	Field Size	Data Format	Content
0	2	Binary	The eventual location of the corresponding field entry in the communication module
2	4	Pointer	Link to the next field in its segment
6	8	Character	Field name
14	2	Binary	Field length minus one
16	2	Binary	Total number of digits or characters in this field
18	2	Binary	Offset to this field within the logical segment
20	2	Binary	Number of decimal places for numeric fields
22	1	Bit	Field type indicator

CHAPTER IX

PRECOMPILER OPTIONS

Run-time options are passed to the precompiler by means of a parameter string specified on the EXEC statement of the invoking JCL. Standard OS parameter conventions apply. Each option has a default, as indicated in Table 10; options may be specified in any order separated by commas. If an option appears more than once, the last specification (scanning left to right) is used. Each option keyword may be abbreviated with any number of characters up to its complete spelling. For the options which may be prefixed by "NO," abbreviations still apply with or without the prefix. A description of each option along with any unique specification rules is shown in Table 10.

TABLE 10
PRECOMPILER OPTIONS

Keyword	Meaning
DECK/NODECK	This option indicates whether a card image version of the PL/I code produced is to be written to file SYSPNCH for punching. Default is NODECK.
GEN/NOGEN	This option indicates whether the PL/I code produced is to be written to file SYSLIN and the communication CSECT is to be written to file SYSOUT. Default is GEN.
INSOURCE/NOINSOURCE	This option indicates whether the input file is to be listed on file SYSPRINT. Default is NOINSOURCE.
LIST/NOLIST	This option indicates whether the PL/I code produced is to be listed on file SYSLIST. Default is NOLIST.
MARGINS(a,b,c)	This option indicates the source margins applicable to the input. All values must be between 0 and 80 inclusive. Only data within the source margin is processed. a - The left margin. Default is 2. b - The right margin. Default is 72. c - The carriage control character position, used when printing the insource. If 0, then single spacing is used. Default is 0.
NUMBER/NONUMBER	This option indicates whether the PL/I code produced should be renumbered, starting with 10 and incrementing by 10 in the sequence area defined by the SEQUENCE option. Default is NONUMBER.
SEQUENCE(x,y)	This option indicates the position of the sequence field in the input record. x - The left margin of the sequence field. Default is 73. y - The right margin of the sequence field. Default is 80.

CHAPTER X

PRECOMPILER OPERATING ENVIRONMENT

The precompiler was written in PL/I and compiled using IBM's PL/I Optimizing Compiler. It is intended to be executed on IBM 370 hardware with access to the dictionary system of Appendix A. Five input files and five output files are used by the precompiler in its processing. Four of the input files are the dictionary's four data sets. These data sets are VSAM files and access to them is through special dictionary service modules. A full description of each of these four dictionary system files can be found in Appendix A. The following table describes the files used by the precompiler. Given are the internal PL/I file names, the associated DDNAME and compiler options, the formats of each file with characteristics when different from the PL/I default, and the usage for each file.

A set of PL/I preprocessor macros are used to generate the three VSAM control blocks within the precompiler program. These blocks are the Access-Method Control Block (ACB), the Request Parameter List (RPL), and the Exit List (EXLST). With these control blocks and an external assembler routine, the precompiler has full access to the NODE data set. Access to the LAT table, HOJ table and the EDGE data sets is through a set of I/O routines, one for each data set. These assembler routines are tailored for the type of requests that are made against their particular data set. All access to the other files used by the precompiler is through standard PL/I I/O.

TABLE 11
INPUT AND OUTPUT FILES

File Name	DDNAME	Compiler Option	Format	Usage
SYSIN	SYSIN	N/A	STREAM, INPUT	The input source program to be precompiled
SYSPRINT	SYSPRINT	INSOURCE	PRINT, LINESIZE(130), VBA, LRECL(135), BLKSIZE(139)	The listing file that contains the header page, insource listing and error messages
SYSLIST	SYSLIST	LIST	PRINT	The listing of the output source pro- gram generated
SYSLIN	SYSLIN	GEN	RECORD, OUTPUT, FB, LRECL(80), BLKSIZE(1680)	The output source program generated
SYSPNCH	SYSPUNCH	DECK	RECORD, OUTPUT, F, LRECL(80)	The to-be-punched form of the source program generated
SYSOUT	SYSOUT	N/A	RECORD, OUTPUT, FB, LRECL(80), BLKSIZE(1680)	The communication module in object form
N/A	LATTABLE	N/A	VSAM, ESDS	The dictionary LAT data set
N/A	NODE	N/A	VSAM, KSDS	The dictionary NODE data set
N/A	EDGE	N/A	VSAM, ESDS	The dictionary EDGE data set
N/A	RNPDDHOJ	N/A	VSAM, ESDS	The dictionary HOJ data set

CHAPTER XI

TESTING AND VERIFICATION

The data base precompiler was implemented in a structured, top down fashion. Therefore, by using "null" routines where routines were not yet implemented, each section being programmed could be tested. With this technique the parameter parsing section was written and tested first. Following that the lexical analysis section was programmed and tested, then the syntactic and semantic routines.

Testing the semantic routines was the most difficult part. Since the Data Base Dictionary System described in Appendix A was not fully implemented, exhaustive system testing was impossible. A sample data definition language and dictionary maintenance subsystem was not developed at all. In light of this, only a small set of test data was loaded into test dictionary data sets to allow the precompiler to test its access and use of the dictionary. Although not a thorough test, this does show the feasibility of such a precompiler as part of a dictionary system.

APPENDIX A

INTRODUCTION

Appendix A is a description of the Data Base Dictionary System. Section I gives an overview of VSAM, IBM's Virtual Storage Access Method. Only terminology and concepts necessary to the reader's understanding of the following system description have been included. Section II is a system overview that discusses the dictionary itself as well as the role played by the precompiler and execution monitor. The third and final section gives detailed record layouts of the data records in each of the four data sets composing the dictionary.

SECTION I

VSAM OVERVIEW

Because our Data Base Dictionary System was implemented on IBM computing hardware, using IBM software for support, it was necessary to choose an access method from those available with current IBM operating systems. Our requirements were quite varied. In addition to direct access by pointer within and between data sets, we needed direct and sequential access by key value. VSAM (Virtual Storage Access Method) was chosen because it supported all our processing needs. The following is a short overview of VSAM and the terminology used when describing our use of it.

VSAM offers two types of data sets, key-sequenced data sets (KSDS) and entry-sequenced data sets (ESDS). The primary difference between the two is the order in which records are stored within them. In a KSDS the records are stored in sequence by the value of a specified key field from each record. Sequential and direct access is possible via this key field. In an ESDS, records are stored without regard to data within the records. The sequence of an ESDS is determined by the order in which records were stored. Physical sequential access is allowed as well as direct access by relative byte within the data set.

Both ESDS and KSDS are actually stored and retrieved in units called control intervals. The total space of a data set is considered to be divided into a continuous set of these control intervals; hence

a data record stored within a control interval can be addressed by its Relative Byte Address (RBA), i.e., offset, in bytes, from the beginning of the data set. We have used these RBA's for our direct pointer implementation both within a data set and between data sets. A complete description of the VSAM access method can be found in appropriate IBM documentation and publications [6].

SECTION II

SYSTEM OVERVIEW

The Data Base Dictionary System was designed to be an administrative aid as well as the source of information used to allow and control access to data in a particular environment. Five different levels of data are recognized as separate entities by the dictionary. These entities are: fields, segments, data bases, programs, and systems. For each of these entities the dictionary maintains information on its characteristics, usage and relationship with other entities. Each entity type is represented as a node in the graphical diagram of the dictionary system (Figure L), and the five nodes have been labeled N1 through N5. All node data, however, is kept in one VSAM key-sequenced data set (KSDS). In addition to the static information about each entity, inter-node relationships are maintained to build levels of data, that is, several fields make up a segment, several segments make up a data base, several data bases may be used by one program, and several programs may belong to one system.

Certain types of information have meaning only as they relate one entity to another; for example, a field's location is significant only as that field relates to a particular segment. This type of relational information is called "edge" data. The four different types of edge data are represented in Figure L by the labels E1 through E4, and are stored in a VSAM entry-sequenced data set (ESDS). An example

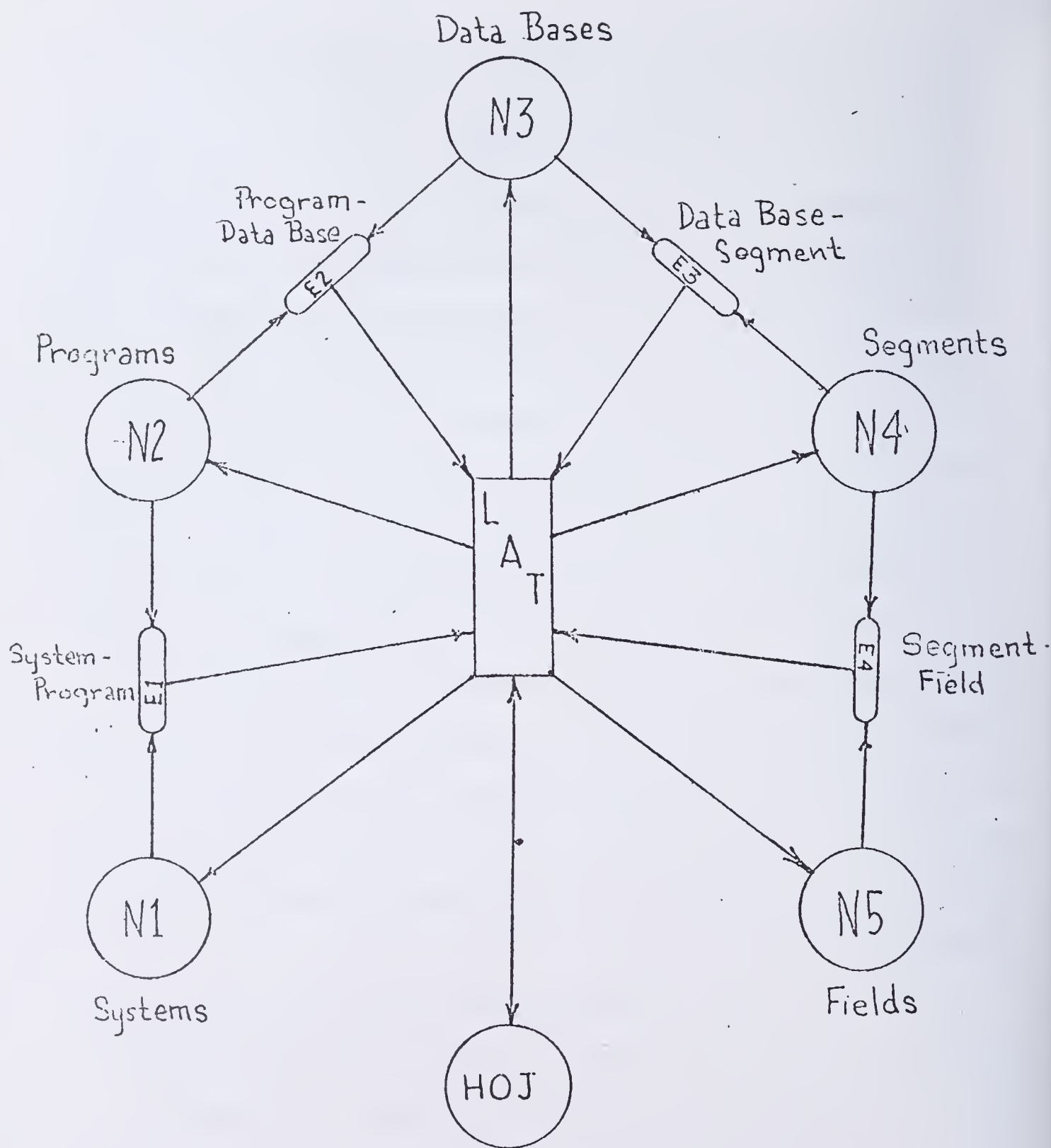


Fig. L.--Logical view of dictionary system

showing how the segment-field edge data exists is given in Figure M. A segment points into the edge data set to the head of a linked list connecting all the edge entries for all the fields in that segment. In like manner, a field points into the edge data set to the head of a list linking all occurrences of that field in the several segments in which it might exist. Each edge entry points to both of the node entries it relates.

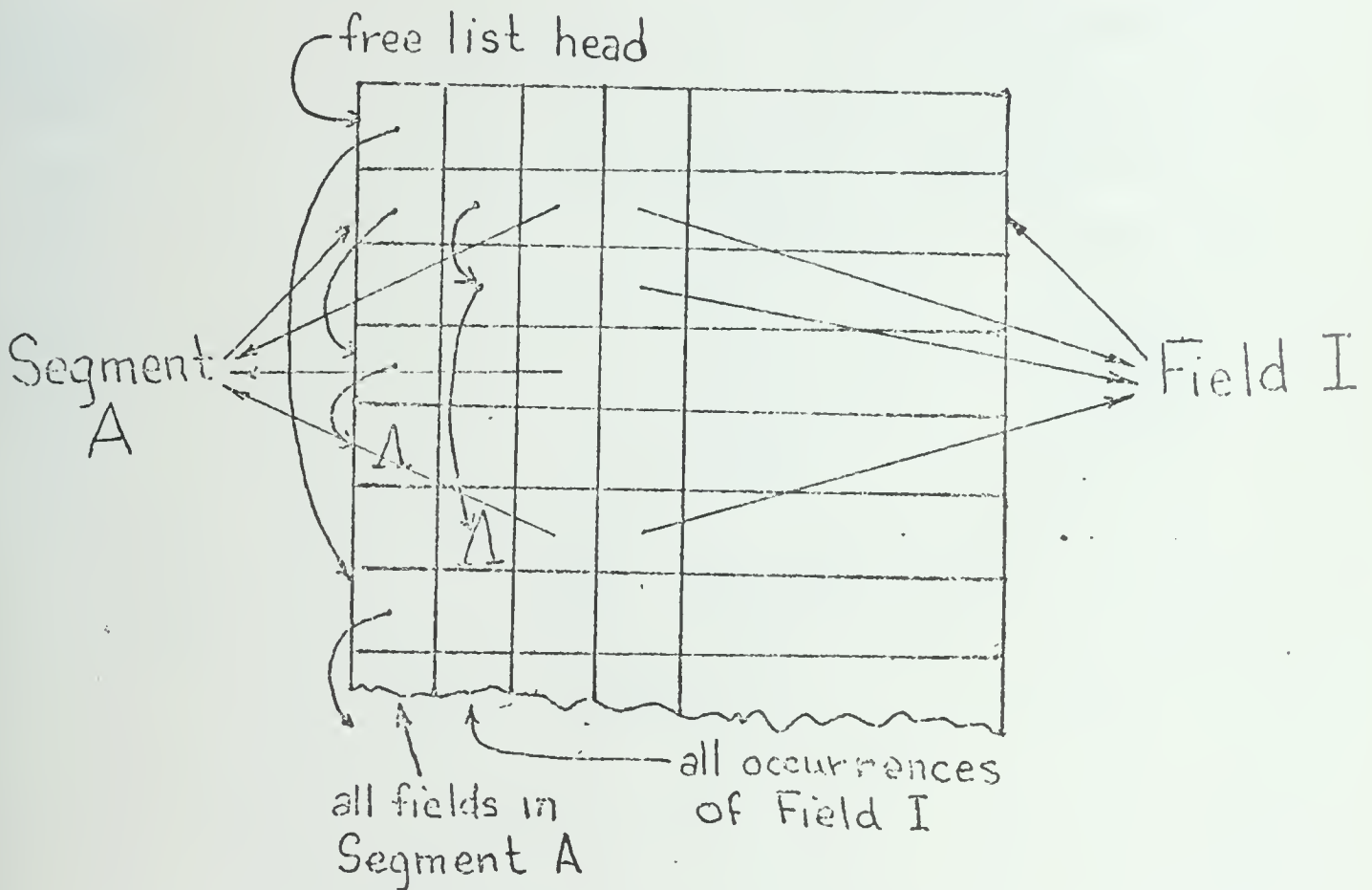


Fig. M.--Example of segment-field edge

Infrequently needed or variable-length information for any of the nodes or edges is kept in the H0J table; this is a VSAM ESDS. This device was chosen to improve efficiency from both the storage and the

processing point of view. The HOJ table allows the data sets containing all of the information pertinent to the node or edge to consist of fixed-length records. Figure N gives a representational view of the HOJ table. A variable number of fixed-length records make up one entry of information. These records are linked, and the node or edge entry referencing the information in the HOJ table contains a pointer to the head of this list.

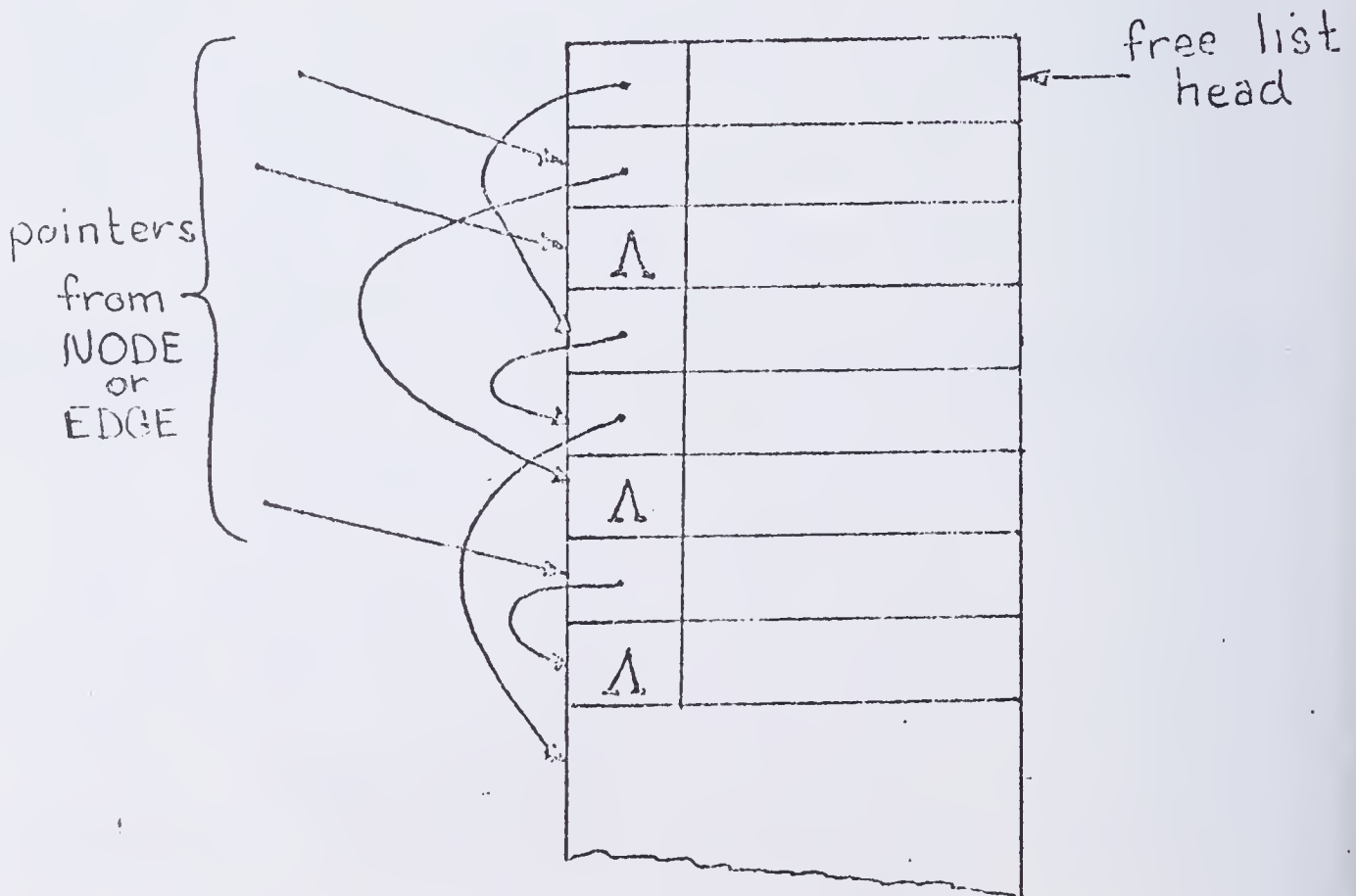


Fig. N.--Logical view of HOJ table

The dictionary uses relative byte addresses (RBA) as direct pointers from one entry to another, the latter being either in the same

data set as the former or in one of the other three data sets composing the dictionary. In VSAM KSDK's the RBA of existing records can change as records are added, changed, or deleted. In order to minimize the effect of this relocation of records in the node data set, an indirect pointer scheme is used. A separate data set, the LAT table, is used to implement these indirect pointers. Figure 0 shows how a node, edge, or HOJ entry points into the LAT table, which in turn points at the target node entry. As RBA's change in the node data set, the corresponding LAT table entry is updated. With this technique, the many pointers that reference a particular entry can be maintained by updating only one indirect pointer.

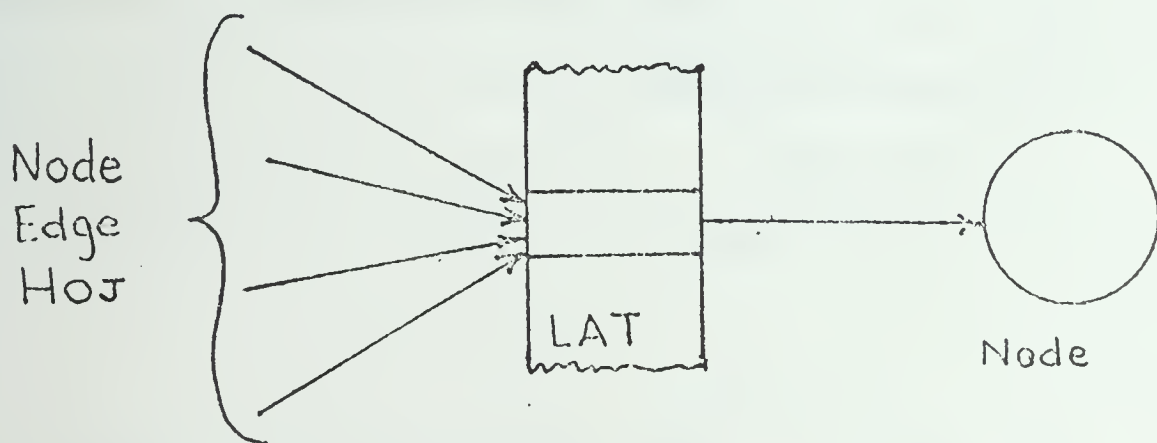


Fig. 0.--Logical view of LAT table

In order to offer control and several other features, the dictionary system has two major subsystems. These are a PL/I precompiler and an execution monitor. Both of these subsystems access the dictionary

data for the information needed to provide various services outlined below.

1. Security enforcement to the field level.
2. A shorthand for some of the control blocks and call statements.
3. The definition of logical segments of data.
4. An interface module for communication with the execution monitor.

The execution monitor acts as an interface between the application program and the data base software, thereby allowing several features not inherent in the data base software to be available. These include:

1. Translation between user defined logical segments and real data segments.
2. Data editing.
3. Data compression and "invisible" fields with default values.
4. Derivable fields.

The concept of a logical segment defined by an application program proves useful in several ways. It first helps clean the users code by deleting filler fields in input/output data structures. It frees the user from being tied to data of specific characteristics, and finally, it allows a program to be desensitized to data at the field level.

SECTION III

DETAILED DESCRIPTION OF THE DICTIONARY DATA SETS

This section provides a detailed record layout of the entire Data Base Dictionary System. As discussed above, the system comprises four separate data sets linked by RBA pointers: the node, edge, LAT, and HOJ data sets.

NODE RECORDS

The node data set is a VSAM/KSDS file whose key is composed of a one-byte type identifier and an eight-byte name, for a total key length of nine bytes. All node records are thirty-eight bytes long. The control interval size is 512 bytes. There are five different types of node records. The fields making up the various node records are explained below in Tables 12 through 16.

TABLE 12

SYSTEM NODE

Decimal Dis- placement	Field Size	Data Format	Content
0	1	Character	"Y" to identify system node
1	8	Character	System name
9	3	Binary	RBA of LAT entry for this node record
12	3	Binary	RBA of HOJ entry for the text string de- scribing this system
15	3	Binary	RBA of the first system/ program edge entry for this system
18	8	Character	Password for this system
27	1	Binary	System type. Possible codes: Bit 0=1 system Bit 1=1 trans- action Bit 2=1 job- stream

TABLE 13
PROGRAM NODE

Decimal Dis- placement	Field Size	Data Format	Content
0	1	Character	"P" to identify program node
1	8	Character	Program name
9	3	Binary	RBA of LAT entry for this node record
12	3	Binary	RBA of HOJ entry for the text string de- scribing this program
15	3	Binary	RBA of the first system/ program edge entry for this program
18	3	Binary	RBA of the first program/ data base edge entry for this program
21	3	Binary	Program input/output area size
24	3	Binary	Program segment search area size
27	1	Binary	Program type. Possible codes: Bits 0-1=00 PL/I Bits 01 assembler 10 COBOL Bit 2 = 0 CMPAT=NO 1 CMPAT=YES
28	3	Binary	Maximum enqueue calls allowed at any one time

TABLE 14
DATA BASE NODE

Decimal Dis- placement	Field Size	Data Format	Content
0	1	Character	"D" to identify data base node
1	8	Character	Data base name
9	3	Binary	RBA of LAT entry for this node record
12	3	Binary	RBA of HOJ entry for text string describing this data base
15	3	Binary	RBA of the first program/ data base edge entry for this data base
18	3	Binary	RBA of the first data base/segment edge entry for this data base
21	3	Binary	RBA of the shared secondary index head node LAT entry
24	3	Binary	RBA of the LAT entry of the next shared secondary index data base node
27	1	Binary	Data base type. Possible codes: Bit 0=1 HSAM Bit 1=1 SHSAM Bit 2=1 HISAM Bit 3=1 SHISAM Bit 4=1 HDAM Bit 5=1 HIDAM Bit 6=1 INDEX Bit 7=1 LOGICAL
28	1	Binary	Physical access method. Possible codes: Bit 0=1 ISAM Bit 1=1 VSAM Bit 2=1 OSAM Bit 3=0 NOPROT 1 PROT

TABLE 14--Continued

Decimal Dis- placement	Field Size	Data Format	Content
29	3	Binary	RBA of the H0J entry de- scribing the randomizing module for this data base RBA of the first H0J entry giving data set group information for this data base
32	3	Binary	

TABLE 15
SEGMENT NODE

Decimal Dis- placement	Field Size	Data Format	Content
0	1	Character	"S" to identify system node
1	8	Character	Segment name
9	3	Binary	RBA of LAT entry for the record
12	3	Binary	RBA of H0J entry for the text string describing this segment
15	3	Binary	RBA of the first data base/segment edge entry for this segment
18	3	Binary	RBA of the first segment/ field edge for this segment
21	3	Binary	RBA of the LAT entry for the physical source seg- ment for this segment
24	3	Binary	RBA of the LAT entry for the physical sibling segment for this segment

TABLE 15--Continued

Decimal Dis- placement	Field Size	Data Format	Content
27	1	Binary	Segment type. Possible codes: Bit 0=0 Non-compressible 1 Compressible Bit 1 Indicates how the pointer segment participates in the concatenated segment being defined =0 Physically =1 Virtually Bit 2 Indicates how the segment pointed at participates in the concatenated segment being defined =0 Physically =1 Virtually Bit 3=1 Key of segment being pointed at is stored in this segment
28	2	Binary	Maximum length of the segment
30	2	Binary	Minimum length of the segment
32	3	Binary	RBA of edge entry for logical source segment
35	3	Binary	RBA of edge entry for destination source segment

TABLE 16
FIELD NODE

Decimal Dis- placement	Field Size	Data Format	Content
0	1	Character	"F" to identify field node
1	8	Character	Field name
9	3	Binary	RBA of LAT entry for this node record
12	3	Binary	RBA of HOJ entry for the text string describing this field
15	3	Binary	RBA of the first segment/ field edge for this field
18	3	Binary	Pointer to field edit information in HOJ
21	3	Binary	Indirect RBA of the generic parent field node for this field
24	3	Binary	Indirect RBA of the next generic sibling field node for this field
27	1	Binary	Field type. Possible codes: Bit 0=1 FLOAT Bit 1=1 FIXED Bit 2=1 CHAR Bit 3=1 PACKED Bit 4=1 ZONED Bit 5=1 /SX Bit 6=1 /CK Bit 7=1 XDFIELD Bits 0-7=1 HEX
28	2	Binary	Field length
30	1	Binary	Decimal places in this field

EDGE RECORDS

The edge data set is a VSAM/ESDS file whose records contain thirty-eight bytes of data within control intervals of 512 bytes each. All access is by direct RBA to the desired record. Initially this data set is completely filled and all the records are linked on a "free list" from which records are made available as they are needed. The different record types and the fields that comprise them are described in Tables 17 through 22 below.

TABLE 17
SYSTEM/PROGRAM EDGE

Decimal Dis- placement	Field Size	Data Format	Content
0	3	Binary	RBA of the next system/ program edge for this system
3	3	Binary	RBA of the next system/ program edge for this program
6	3	Binary	RBA of the LAT entry for the system node partici- pating in this relation- ship
9	3	Binary	RBA of the LAT entry for the program node partici- pating in this relation- ship

TABLE 18
PROGRAM/DATA BASE EDGE-FIRST PCB ENTRY

Decimal Dis- placement	Field Size	Data Format	Content
0	3	Binary	RBA of the next program/ data base edge for this program, i.e., next PCB for this program
3	3	Binary	RBA of the next program/ data base edge for this data base
6	3	Binary	RBA of the LAT entry for the program node partici- pating in this relation- ship
9	3	Binary	RBA of the LAT entry for the data base node par- ticipating in this relationship
12	1	Binary	PCB type. Possible codes: Bit 0 =0 single positioning 1 multiple positioning Bits 1-3=000 processing option G 001 processing option I 010 processing option R 011 processing option D 100 processing option A 101 processing option L Bit 4 =1 processing option E Bit 5 =1 processing option S

TABLE 18--Continued

Decimal Dis- placement	Field Size	Data Format	Content
			Bit 6 =1 processing option P
			Bit 7 =1 processing option 0
13	2	Binary	This is the length of the longest concatenated key in this PCB
15	3	Binary	If a secondary processing sequence is used, this is the RBA of the LAT entry for the secondary index data base
18	10	Binary	First SENSEG entry, see TABLE 20
28	8	---	Unused
36	3	Binary	RBA of the next edge record for this PCB

TABLE 19

ADDITIONAL PCB ENTRIES

Decimal Dis- placement	Field Size	Data Format	Content
0	10	Binary	A SENSEG entry, TABLE 20
10	10	Binary	A SENSEG entry, TABLE 20
20	10	Binary	A SENSEG entry, TABLE 20
30	6	---	Unused
36	3	Binary	RBA of the next edge record for this PCB

There will be one SENSEG for each sensitive segment in the PCB being described. The format of a SENSEG entry is shown in Table 20.

TABLE 20
SENSEG EDGE ENTRY

Decimal Dis- placement	Field Size	Data Format	Content
0	3	Binary	RBA of data base/segment edge for this segment
3	3	Binary	RBA of data base/segment edge for the parent seg- ment of this segment in this hierarchy
6	1	Binary	Processing options for this segment. Bit 0 unused Bits 1-3=000 G 001 I 010 R 011 D 100 A 101 L Bit 4=1 E 5=1 S 6=1 P 7=1 K

TABLE 21
DATA BASE/SEGMENT EDGE

Decimal Dis- placement	Field Size	Data Format	Content
0	3	Binary	RBA of the next data base/segment edge for this data base
3	3	Binary	RBA of the next data base/segment edge for this segment
6	3	Binary	RBA of the LAT entry for the data base participating in this relationship
9	3	Binary	RBA of the LAT entry for the segment participating in this relationship
12	1	Binary	Pointers used: Bit 0=0 SNGL 1 DBLE Bit 1=0 VIRTUAL 1 PHYSICAL Bits 2-4=001 HIER 010 HIERBWD 011 TWIN 100 TWINBWD 101 NOTWIN 110 LTWIN 111 LTWINBWD Bit 5=1 LPARENT Bit 6=1 counter present Bit 7=1 paired
13	3	Binary	RBA of the LAT entry for the parent of this seg- ment in this data base hierarchy
16	3	Binary	RBA of the HOJ entry con- taining LCHILD data for this segment
19	3	Binary	RBA of the HOJ entry for the data set group in which this segment belongs

TABLE 21--Continued

Decimal Dis- placement	Field Size	Data Format	Content
22	3	Binary	RBA of the HOJ entry containing the logical parent data, if applicable
25	4	Binary	Frequency of occurrence of this segment in hundredths
29	1	Character	The insert rule for this segment
30	1	Character	The delete rule for this segment
31	1	Character	The replace rule for this segment
32	1	Character	The nonunique sequence where rule for this segment

TABLE 22
SEGMENT/FIELD EDGE

Decimal Dis- placement	Field Size	Data Format	Content
0	3	Binary	RBA of the next segment/ field edge for this segment
3	3	Binary	RBA of the next segment/ field edge for this field
6	3	Binary	RBA of the LAT entry for the segment node partici- pating in this relation- ship
9	3	Binary	RBA of the LAT entry for the field node partici- pating in this relation- ship
12	1	Binary	Field type. Possible codes: Bits 0-1=10 unique key field 11 multiple- valued key field 00 not key field
13	3	Binary	RBA of HOJ security records for this field
16	3	Binary	RBA of HOJ XDFLD records, if applicable
19	2	Binary	The relative field posi- tion of this field in this segment
21	3	Binary	RBA of HOJ default value or derivable field data, if applicable
24	1	Binary	Secondary index XDFLD constant value
25	1	Binary	Secondary index NULLVAL value
26	8	Character	Name of secondary index exit routine

H0J RECORDS

The H0J data set is a VSAM/ESDS file that is used as a secondary storage area for infrequently needed or variable-length data. The thirty-eight byte logical records are stored in 512 byte control intervals. All five of the node entities make use of the H0J table as well as several of the edge entries. Textual descriptions, edit information, and default values are examples of the types of data stored in the H0J table. Each logical record has room for thirty-four bytes of data and four bytes of control information. A free list, whose head is the first H0J record, is maintained in order to link all unused records together. Table 23 below shows the layout of a H0J record.

TABLE 23
SAMPLE H0J RECORD

Decimal Dis- placement	Field Size	Data Format	Content
0	1	Binary	Length of data portion
1	3	Binary	Next record RBA
4	34	Mixed	Data (variable)

The data portion of each H0J record is different depending on the type of record. Much of the information kept here is used to generate IMS control blocks. [7] can be consulted as to the meaning of many of the fields. The different usages and layouts of the data portion of a H0J entry, which may be more than one H0J record, is shown in Tables 24-28.

TABLE 24
RANDOMIZING MODULE HOJ DATA

Decimal Dis- placement	Field Size	Data Format	Content
0	8	Character	Module name
8	4	Binary	Number of root anchor points
12	4	Binary	Maximum relative block number
16	4	Binary	Maximum number of bytes of a data base record stored in the root addressable area

TABLE 25
EDIT/VERIFICATION HOJ DATA

Decimal Dis- placement	Field Size	Data Format	Content
0	1	Binary	Type of verification. Possible codes: Bit 1=1 range of possible values 2=1 list of possible values
1 3	2 --	Binary ---	Number of values The two values or the list of possible field values. The format and length match the field characteristics

TABLE 26
XDFIELD HOJ DATA

Decimal Dis- placement	Field Size	Data Format	Content
0	3	Binary	RBA of source segment
3	3	Binary	RBA of search field one
6	3	Binary	RBA of search field two
9	3	Binary	RBA of search field three
12	3	Binary	RBA of search field four
15	3	Binary	RBA of serach field five
18	3	Binary	RBA of subsequence field one
21	3	Binary	RBA of subsequence field two
24	3	Binary	RBA of subsequence field three
27	3	Binary	RBA of subsequence field four
30	3	Binary	RBA of subsequence field five
33	3	Binary	RBA of duplicate data field one
36	3	Binary	RBA of duplicate data field two
39	3	Binary	RBA of duplicate data field three
42	3	Binary	RBA of duplicate data field four
45	3	Binary	RBA of duplicate data field five

TABLE 27
DATA SET GROUP HOJ DATA

Decimal Dis- placement	Field Size	Data Format	Content
0	6	Character	Data set group name
6	8	Character	DDNAME one
14	8	Character	DDNAME two or overflow DDNAME
22	2	Binary	Block factor one
24	2	Binary	Block factor two
26	2	Binary	Size/record length one
28	2	Binary	Size/record length two
30	1	Binary	Scan limit
31	1	Binary	Free block frequency factor
32	1	Binary	Free space percentage factor
33	1	Binary	Model and device type. Possible codes: Bit 0=1 2314 Bit 1=1 2305 Bit 2=1 2319 Bit 3=1 3330 Bit 4=1 3340 Bit 5=1 2400 Bit 6=1 3400 Bit 7=0 2305 model 1 or 3330 model 1 1 2305 model 2 or 3330 model 11

TABLE 28
LOGICAL CHILD HOJ DATA

Decimal Dis- placement	Field Size	Data Format	Content
0	3	Binary	RBA of data base/segment edge for the logical child segment
3	1	Binary	Pointer type. Possible codes: Bit 0=1 SNGL Bit 1=1 DBLE Bit 2=1 NONE Bit 3=1 INDX Bit 4=1 SYMB
4	3	Binary	RBA of data base/segment edge for paired segment
7	3	Binary	RBA of segment/field edge for the index field
10	1	Character	Insert rules, either "F", "H", "L"

In addition to the above uses of the HOJ records, there are three other uses that do not lend themselves to tabular description. They are textual descriptions, security records, and the default value/derivable field records. In the textual records the character string description is packed into as few records as possible. Security records relate to a segment/field edge entry and are made up of a series of four byte entries. Each entry gives the RBA of a program node corresponding to a program that has access to the field and an indication as to the type of access. If the record is a default value entry, then it contains the default value of a field packed into as few HOJ records as possible.

For derivable fields, the module name and the RBA pointer to the argument(s) passed to that module are stored.

LAT RECORDS

The LAT data set is a VSAM/ESDS file which has a record length of four bytes, containing only a type byte and the RBA of a node record. This allows modification of the placement of the node records without changing all pointers to these entries, since all pointers point through the LAT. Only this single RBA need be updated. Table 29 below describes the layout of the LAT record.

TABLE 29
LAT RECORD

Decimal Dis- placement	Field Size	Data Format	Content
0	1	Binary	Type of entry pointed at. Possible codes: Bit 1=1 field Bit 2=1 segment Bit 3=1 data base Bit 4=1 program Bit 5=1 system Bit 6=1 generic head Bit 0-7=0 free
1	3	Binary	RBA of entry pointed at

REFERENCES

1. Cohen, Leo J. Data Base Management Systems: A Critical and Comparative Analysis. Performance Development Corporation, Trenton, New Jersey, 1973.
2. Nerad, Richard A. "Data Administration as the Nerve Center of a Company's Computer Activity," Data Management, vol. 11, no. 10, October 1973, 26-31.
3. "The Data Dictionary/Directory Function," EDP Analyzer, vol. 12, no. 11, November 1974, 1-13.
4. Information Management System Virtual Storage (IMS/VS), General Information GH20-1260, IBM Corp., White Plains, New York, March 1974.
5. Information Management System Virtual Storage (IMS/VS), Application Programming Reference Manual SH20-9026, IBM Corp., White Plains, New York, August 1974.
6. OS/VS Virtual Storage Access Method (VSAM), Programmer's Guide GC26-3818, IBM Corp., White Plains, New York, May 1973.
7. Information Management System Virtual Storage (IMS/VS), Utilities Reference Manual SH20-9029, IBM Corp., White Plains, New York, August 1974.

BIOGRAPHIC DATA ET	1. Report No. UIUCDCS-R-76-798	2.	3. Recipient's Accession No.
	Title and Subtitle The Precompiler Component of a Data Base Dictionary System		5. Report Date May 1976
6.		8. Performing Organization Rept. No. UIUCDCS-R-76-798	
Author(s) Michael Jason Huggins		10. Project/Task/Work Unit No.	
Performing Organization Name and Address Department of Computer Science University of Illinois at Urbana-Champaign Urbana, Illinois 61801		11. Contract/Grant No.	
Sponsoring Organization Name and Address Department of Computer Science University of Illinois at Urbana-Champaign Urbana, Illinois 61801		13. Type of Report & Period Covered Master of Science Thesis	
14.			
Supplementary Notes			
Abstracts With the advent of large, general purpose data base systems, several desirable information processing theories have now been implemented. These include advances in the areas of data independence, data sharing, data security, and control. While facilities to take advantage of these concepts have been implemented to varying degrees, much of the control needed to administer their use is not inherent in the data base software itself. To meet this need, the role of data base administration has emerged. While data base administration is finding its place in data processing structures, much work is being done to provide it with the tools needed to manage and control the data.			
Key Words and Document Analysis. 17a. Descriptors Data Dictionary precompiler			
Identifiers/Open-Ended Terms			
COSATI Field/Group			
Availability Statement Release Unlimited		19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 79
		20. Security Class (This Page) UNCLASSIFIED	22. Price -----

104 8 1078



UNIVERSITY OF ILLINOIS-URBANA

510.84 IL6R no. C002 no. 794-799(1976)
Statistical estimation of throughput and



3 0112 088402711